CEWES MSRC/PET TR/98-31

MPICH on the Cray T3E

by

L. Shane Hebert
Walter G. Seefeld
Anthony Skjellum

05h00798

Work funded by the DoD High Performance Computing Modernization Program CEWES Major Shared Resource Center through

Programming Environment and Training (PET)

Supported by Contract Number: DAHC 94-96-C0002
Nichols Research Corporation

MPICH on the Cray T3E

L. Shane Hebert Walter G. Seefeld Anthony Skjellum
{shane,walt,tony}@aurora.cs.msstate.edu
High Performance Computing Laboratory
Engineering Research Center
Mississippi State University
Mississippi State, MS 39762
http://www.erc.msstate.edu/labs/hpcl
March 27, 1998

Abstract

The authors describe their efforts to support the ubiquitous MPI programming model, based on the MPICH 1.1 implementation, on the Cray T3E. Discussion of porting issues, performance issues, and protocol decisions are offered. Future work opportunities and challenges are also reported.

Keywords: parallel computing, message passing, Cray T3E, MPI, MPICH

1 Introduction

Message passing has been used for many years to program multicomputers and clusters of workstations. In recent years, the Message Passing Interface (MPI) Standard [1] Application Programmers' Interface (API) has become the de facto standard for writing portable message-passing programs. A portable, public implementation of this standard, MPICH [2], written jointly by Argonne National Laboratory and Mississippi State University (MSU), has been ported to a wide variety of computing systems. Among these ports is one for the Cray T3D which was first written by Brightwell from MSU [3] and later modified to conform to MPICH's second generation Abstract Device Interface (ADI-2) by Shane Hebert. Although the Cray T3D and Cray T3E are similar machines and are source-code portable, there are optimizations in the T3E architecture that can be used to write a more optimal port of that device code for the T3E.

1.1 MPI

The MPI Standard, defined by the MPI Forum, has become the de facto standard for writing message-passing programs [1]. The standardization effort was started in 1993 and the standard was approved by the MPI Forum in 1994. Since then, many vendors of multicomputers, shared-memory computers, and clusters have provided proprietary implementations of MPI for their systems. Several public implementations are also available [4, 5].

MPI allows programmers to write portable parallel message passing program code. Because the programs are portable, software writers can target multiple platforms easily, most often simply by recompiling the source code on the target platform. This allows MPI applications to be useful long after the platform that was originally used to develop the code has become obsolete. Applications written on one system can be easily ported to a new computing system that is purchased to replace an obsolete system. The program can be ported even if the newer system is not otherwise compatible with the older system in other ways such as interconnection infrastructure or binary compatibility.

1.2 MPICH

MPICH is a public, portable implementation of the MPI Standard that was developed in parallel with the standardization effort and was originally released at the same time as the standard. It was written with join effort from Argonne National Laboratory and Mississippi State University. Since then, it has become the starting point for many vendors' implementation efforts and has also been ported to many of those same platforms. MPICH is available for public download. As a consequence, it has been used by many groups, both research and commercial, to port an MPI library to a wide variety of computing systems. The port of interest for this project is the MSU Cray T3D port with its later modifications.

1.3 Cray T3E

The T3E is a recent addition to a family of distributed shared-memory multicomputers manufactured by Cray [6]. This multicomputer is based on the commodity Alpha 21164A microprocessor first manufactured by Digital Equipment Corporation (DEC) [7]. Each node module of the T3E contains four microprocessors, each with its own local bank of DRAM and high-speed network direct memory access (DMA) interconnect. The node modules of the T3E are connected by a toroidal topology. This allows multiple paths to nodes for redundancy and congestion relief. The torus links provide a theoretical bidirectional peak of 480 Mbyte/sec bandwidth between node modules [6]. Communication between and among nodes is accomplished by using distributed shared-memory primitives to push and pull data from one node's memory into another node's memory using the high-speed DMA hardware of the interconnect.

2 Porting MPICH to the Cray T3E

The Cray T3D and the Cray T3E are nearly source-code portable. This means that, with the exception of a few library calls, programs that can be compiled and run on a Cray T3D can be recompiled and run on a Cray T3E with little effort. Consequently, after omitting some T3D specific code, the existing T3D device [1] works well on the T3E. However, architectural optimizations can be used to improve the T3E device. In order to explain the T3E device code, a comparison of the T3D to the T3E is required.

2.1 Comparing the T3D and T3E

There are three major optimizations to the T3D distributed shared-memory architecture in the T3E that can be advantageous to use in the T3E device code. The three deficiencies of the T3D are discussed in detail by Brightwell [3], namely asymmetry of data transfer performance, address validation, and cache-coherency. Each of these is summarized here.

The Cray T3D and T3E computers are both distributed shared-memory computers. To communicate between processing nodes, the shared-memory API (`shmem` library) is used to explicitly transfer regions of memory from one node to another. The `shmem` API contains two types of functionality that are of main concern for the MPICH device code — push and pull. The push functionality is accessible through two functions: `shmem_put` and `shmem_put32`. These two functions perform nearly identical tasks. The difference between the two is the granularity of the addressing and data size handled by the function. The `shmem_put` function requires that the data to be sent be aligned on a 64-bit word boundary and be a multiple of 64-bits in length. The `shmem_put32` function requires only that the data be aligned on a 32-bit word and be a multiple of 32-bits in length. Analogously, the pull functionality also exists in the 64-bit and 32-bit varieties. These func-

[1] The MPICH source code is divided into several layers for portability. The topmost layers of the software are portable across most systems but the bottom-most layer, the device layer, is specific to each particular architecture.

tions are `shmem_get` and `shmem_get32`. There are other functions available as well to handle short, long, longlong, double, long double, 128-bit, and complex data types. We found these functions to be of similar performance to the simple 32- and 64-bit varieties so we do not discuss them further here.

From empirical testing, the Cray T3D has an asymmetry in these two modes of data movement. Out of a theoretical maximum bandwidth of 130 Mbytes/second, the tested push bandwidth of this system is 110 Mbytes/second. However, the tested pull bandwidth is less than one-third of the tested push bandwidth at 30 Mbytes/second. The MPICH Cray T3D device can achieve 107 Mbytes/second using the three-phase push protocol. The T3E, on the other hand, is much more symmetric. The pull functionality is nearly the same bandwidth as the push functionality. Both supposedly can achieve near the peak 480 Mbytes/sec of the interconnection network. However, in practice, this is not the case as shown in Figures 1 and 2. We will discuss these figures later.

The second obstacle to overcome on the T3D involves the virtual memory subsystem of the platform. A design limitation of the T3D does not allow a process on one node to access address regions in another node unless the memory locations in question are mapped into the virtual memory space of both processes. The obvious way to avoid this problem is to use the shared memory allocation function `shmem_alloc` which enforces the design by mapping the shared memory to the same virtual memory address region for all the processes. This seems to solve the problem but, for MPI, this is a considerable hurdle to overcome because user data in MPI is not required to be in memory allocated from this special memory. The operating system of the T3E, however, does not require that the target address region be mapped into the local process space in order to use either the push or the pull data movement functionality. This enhances performance in itself.

The last obstacle is the matter of cache-coherence. The T3D does not provide cache-coherence for the processor for remote memory operations. This requires the implementation to manually invalidate and flush the cache or to simply turn the cache off completely. The T3E, on the other hand, does provide cache-coherency. This is again a performance boost for MPI applications.

2.2 Changes to the T3D Device Code

The improvements in the Cray T3E architecture allow several direct improvements to the T3D device code that both decrease latency and allow the capture of a higher percentage of the peak bandwidth of the T3E.

Because of the asymmetry in the push and pull modes of the T3D, the original and second implementations of the MSU Cray T3D MPICH device code use a three-phase push protocol for messages larger than 256 bytes [3]. These phases are the basic request-to-send/clear-to-send (RTS/CTS) protocol. In this protocol, the sender first notifies the receiver that a message is available for sending (RTS). The receiver later notifies the sender when the data for that message can be sent (CTS). Afterwards, the sender can then push the data directly into the user's buffer to complete the third phase.

For a distributed shared-memory architecture, this protocol can include some optimizations to simplify protocol demultiplexing. For example, the CTS packets and the actual transmission of the data do not have to be in the form of seperate packets over the communication network. The receiver can simply write to specified locations the information that is required to complete the data transfer operation. The sender can examine this information without protocol overhead and complete the sending of the data by directly placing the data in the desired buffer on the receiving node. This satisfies the MPI Progress Rule [8] in that the transmission of data can occur even in the absense of calling the `MPI_Wait` and `MPI_Test` function calls.

From empirical testing, we found that there is an asymmetry in the push and pull modes of the T3E as well. However, as Figures 1 and 2 show, our bandwidth test results using various `shmem` primitives available on the T3E found that the pull mode has higher bandwidth than the push mode, opposite of the bahavior of the T3D. We found the peak obtainable bandwidth using the push mode is 299 Mbytes/sec (Figure 1) and the pull mode is 327 Mbytes/sec (Figure 2) . This performance difference may be attributed to cache-coherency protocols used on the T3E nodes. As each block of data is written during the push from the source, cache-coherency on the target must be preserved piecemeal. Using the pull method, the memory areas that must be invalidated are known at the beginning of the function call and all the cache-coherency issues can be handled in one operation before any data is transferred. Although this asymmetry was not taken into account in the original design of the port, it works in our favor to enforce the original design of using the pull mode on the T3E for MPICH message passing.

We performed experiments with many of the various `shmem` functions and found that within the functionality of each mode, the performance differences between any two of the functions that are differentiated by the address alignment and transfer granularity was less than 1%. This enabled us to use the 32-bit granularity functions to minimize the penalties for misaligned transfers and limit the penalties to only transfers that are not 32-bit aligned and multiples of 32-bits in length. These requirements should encompass the majority of MPI messages in most applications. However, minimizing these penalties is an area for future work.

The behavior of the push and pull models of data movement on the T3E allow a much more simple protocol design that incorporates both the push and the pull models for data delivery. On the T3D, we could only use the push model in order to achieve acceptable performance. In the new model for the T3E, the RTS is given to the receiving process which then pulls the data

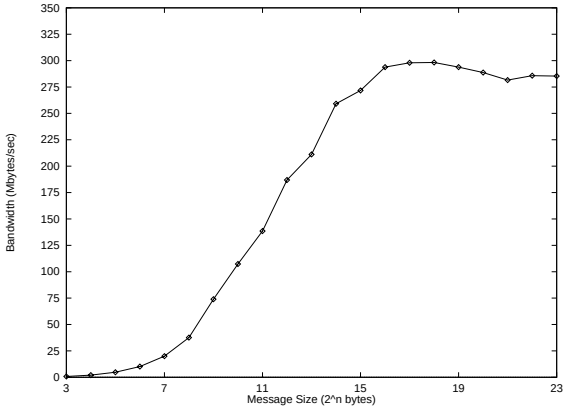

Figure 1: Cray T3E Push Mode Bandwidth as a function of Message Size

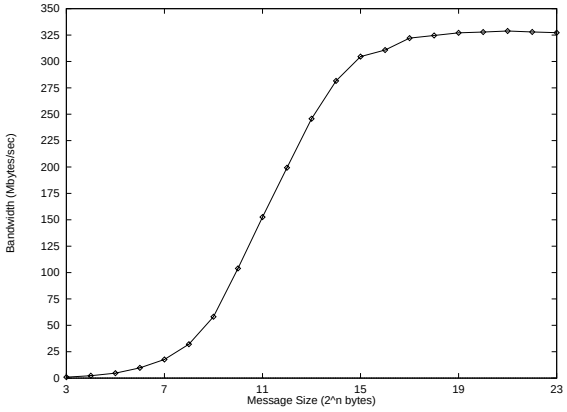

Figure 2: Cray T3E Pull Mode Bandwidth as a function of Message Size

into the user buffer without the requirement of the additional CTS to allow the sender to push the data. Even ignoring the additional bandwidth capability of the T3E network, this simplification of the protocol has both lowered the latency and increased the bandwidth of messages larger than 256 bytes. Also as a result of this simplification, the progress engine of the T3E port to MPICH is much simpler than the progress engine of the T3D port. This new progress engine for the library incurs less overhead to complete data transfers, thus lowering latency even more.

The obstacle of address validation does not initially seem to be a problem because the shared memory allocation routine will ensure that memory is mapped ideally and is readily

usable. However, MPI does not require that all the memory that the user desires to use be allocated in this way. A user may use memory allocated in any way, even from the heap by a call to `malloc`. This poses an interesting problem for the device implementation. The solution to this problem is to trick the operating system into mapping the remote memory address range into the local process when the target address is not already mapped into the local process virtual address space. This requires that, every time data is to be sent, the local process must ensure that the target address range is mapped into the local address space. If the memory is not already mapped into the process space, the local process must force the mapping to occur before the transmission of the data and, after the data is sent, it must unmap that range. The MSU Cray T3D port uses a trick with the process stack to map the target address range into virtual memory when necessary [3]. This, of course, is a penalty to every message greater than 256 bytes in length. The optimization here is to avoid the calls to check the address space mapping and thereby avoiding the calls and overhead to actually map the memory region into the local process space that would be required on the T3D.

The last issue of cache-coherency is completely negated by the cache-coherency present in the T3E. On the T3D, in order to ensure that the processor had copies of the correct data in its L1 cache, the cache could either be manually managed using flush and invalidate calls or it could be simply disabled using system calls. When the programmer issues an MPI call to send data, the MPI send function must ensure that the data being sent is actually flushed out of the cache and into the appropriate memory locations in main memory. Because of the cache-incoherent behavior of the computer, without the flush, it would be possible for the receiver of that data to receive stale or invalid data from the sender. Similarly on the receive side of the operation, every MPI receive function must invalidate the address range of the desired data location or risk the chance of the cache holding data from previous computations which would not be updated by the reception of the data. These issues require the overhead of cache management calls in each and every send and receive call in the MPI library. This penalty is exacted even if it is not needed because the library writer can never be sure if the specified data address locations are cached or not. Of course, if the second option is taken - disabling the cache - the issue is resolved at the expense of overall program execution speed. Simply stated, the code that ensures that caches are flushed or invalidated at the appropriate times on the T3D is not necessary on the T3E and can be omitted from the T3E device code.

3 Performance

The performance for an MPI implementation is typically characterized by its latency and bandwidth measurements. The latency for a zero-length message is a basic measure of overhead in the MPICH port. A message of zero bytes consists of only the MPICH packet header and no data. The time for this packet with no data to be sent from the user code of the sending process and be recognized by the user code of the receiving process is measured from just before the `MPI_Send` function is called in the sending process until the `MPI_Recv` function returns in the receiving process. The program that is traditionally used to measure latency is called "ping-pong" because it bounces a message back and forth a number of times from the one process to the other. The latency calculation is measured by this program as being half of the average time taken for a zero-byte message to make the round trip. The latency for short messages on the T3E is shown in Figure 3. As this graph shows, the latency for a zero-length message for our port is 7 microseconds.

The bandwidth test is very similar except that it bounces messages of increasing size between two processes. The average time taken for a message to bounce between processes some number of times divided by the num-

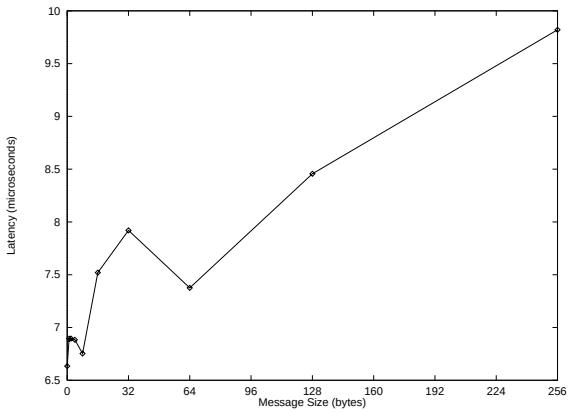

Figure 3: MSU Cray T3E MPICH Latency as a function of Message Size

ber of bytes sent in each bounce, twice the buffer size, is considered to be the average bandwidth for that message size. The bandwidth for our port is shown in Figure 4. Traditionally, the asymptotic bandwidth of the port is used to represent this performance measure. For our port, the asymptotic bandwidth is 321 Megabytes/sec. This falls short of the stated 480 Mbytes/sec peak performance of the T3E [6] by 33%. However, as Figure 2 shows, the maximum bandwidth that we could obtain was 327 Mbytes/sec. Using this empirically obtained peak, we achieved 98% of the obtainable bandwidth of the platform. The MPICH library computation overhead accounts for the other 2%.

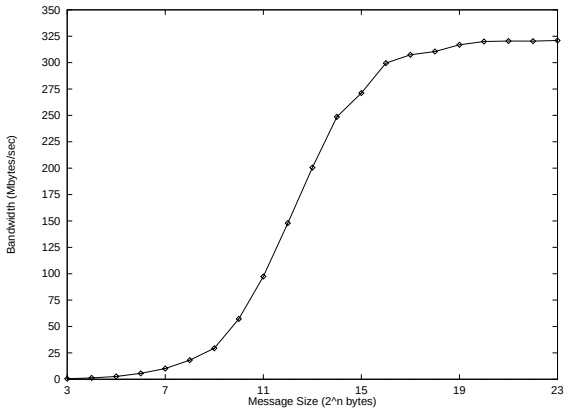

Figure 4: MSU Cray T3E MPICH Bandwidth as a function of Message Size

4 Conclusions

The MPICH device code that targets the Cray T3E can be optimized because the platform corrects some of the deficiencies that are present in the hardware and operating system of the Cray T3D. The protocol for longer messages can be simplified, which means there is less overhead for sending and receiving messages. The code in the T3D device that was needed to work around the addressing and cache coherency deficiencies of that platform is no longer required for the T3E. The device code for the T3E becomes the idealistic device code for the T3D. Of course, the added performance of the T3E interconnection network increases overall performance, but the simplification and optimization of the protocol and progress engine improve the efficiency of the port.

This port also obeys the progress rule of the MPI Standard [8] more closely than other ports. Because the receiver of the message actually initiates and completes the data transfer, computation can truly overlap communication on the send side of the message. The receiver, however, has more relaxed conformance of the progress rule because MPICH library code must be executing before data is actually transferred. Still, progress can be made even without a specific call to `MPI_Wait` or `MPI_Test`, even on the receiving process.

In actuality, these optimizations also add another, possibly overlooked, benefit to the T3E device code. Because the message passing protocol is greatly simplified and because some of the code in the T3D device was in place to work around architectural deficiencies, the T3E device code is smaller and less complicated than the T3D device code. This means that there is less code to write and the code that is written is easier to debug and maintain for the T3E with the added benefit of lower overhead, higher performance message passing code. This is a win-win situation for the developers, maintainers, and ultimately, the users of the MSU Cray T3E port for MPICH.

The MSU Cray T3E MPICH port

is available via WWW web page at http://www.erc.msstate.edu/mpi/mpich-t3e.html or by contacting the authors via email.

5 Future Work

For MPICH, the first, most direct optimizations are to the point-to-point communication infrastructure which are detailed here. In MPICH, the collective communication functionality is implemented algorithmically using the point-to-point functionality. With these optimizations, collective communication should also see increases in performance. However, the Cray T3E also offers native collective communication primitives with optimizations over those of the T3D. Future work on the T3E could include optimizing the collective communication routines to use the high-performance native collective communication primitives whenever possible.

Another area for future work is optimization of the MPI broadcast functionality for the T3E's torus interconnect. Since the T3E's torus interconnect is actually a 3-D mesh, some performance gains may be realized. Work in this area has already been published and can be used for this effort [9, 10].

As mentioned before, there are penalties for MPI messages that are not aligned on 32-bit addresses and for those that are not multiples of 32-bits in length. There are techniques that can be used to minimize these penalties which are not included in this port. Future work could be to optimize these types of messages as well.

Extensions to the MPI Standard [1] included in the MPI-2 Standard [11] are ideally suited to distributed shared-memory architectures. However, the MPICH source base with the device code reported here is not a direct path to MPI-2. The MPI-2 Standard incorporates several functionality chapters that would require major modifications to the entire MPICH source base. This effort would best be served by creating a new source base but in either case, would require an extensive design effort and implementation team to achieve good performance and minimize complexity of the code.

6 Acknowledgements

This work was supported in part by a grant of HPC time from the DoD HPC Modernization Program.

References

[1] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, 1994. http://www.mcs.anl.gov/mpi.

[2] Argonne National Lab and Mississippi State University. MPICH: A Portable Implementation of MPI. Software, April 1997. http://www.mcs.anl.gov/mpi/mpich.

[3] Ron Brightwell and Anthony Skjellum. MPICH on the T3D: A Case Study of High Performance Message Passing. In *Proceedings of the Second MPI Developer's Conference*, pages 2–9, July 1996.

[4] Ohio Supercomputing Center. LAM. Software, 1996. http://www.osc.edu/lam.html.

[5] EPCC. CHIMP. Software, 1996. ftp://www.epcc.ed.ac.uk/chimp/release.

[6] The Cray T3E Series: The Perfect System for Highly Scalable Applications. WWW Page, 1997. http://www.cray.com/products/systems/crayt3e.

[7] Digital Equipment Corporation. WWW Pages, 1998. http://www.digital.com.

[8] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, 1994. Section 3.7.4.

[9] M. Barnett, D. Payne, R. van de Geijn, and J. Watts. Broadcasting on meshes with wormhole routing. *Journal of Parallel and Distributed Computing*, 35(2):111–122, 1996. http://www.cs.utexas.edu/users/rvdg/papers/meshbc.ps.

[10] Jerrell Watts and Robert van de Geijn. A pipelined broadcast for multidimensional meshes. *Parallel Processing Letters*, 5(2):281–292, 1995.

[11] MPI-2 Forum. *MPI-2: Extensions to the Message Passing Interface*, 1997. http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html.